**Sprawozdanie z ćwiczenia nr 2**

**Temat ćwiczenia:** Budowa i działanie sieci jednowarstwowej

Celem ćwiczenia jest poznanie budowy i działania jednowarstwowych sieci neuronowych oraz uczenie rozpoznawania wielkości liter.

1. **Syntetyczny opis budowy sieci jednowarstwowej**

Działanie sieci jednowarstwowej polega na tym, że pojawienie się określonego wektora wejściowego X powoduje powstanie sygnałów wyjściowych 𝑦𝑚 na wszystkich neuronach wchodzących w skład rozważanej warstwy. Oczekujemy przy tym maksymalnego sygnału wyjściowego 𝑦𝑚 na tym neuronie, którego wektor wag 𝑊(𝑚) najbardziej przypomina X. Sieć tego typu może więc rozpoznawać k różnych klas obiektów, gdyż każdy neuron zapamiętuje jeden wzorcowy obiekt, na którego pojawienie się jest „uczulony”.

O tym, do której klasy należy zaliczyć aktualnie pokazany obiekt, decyduje numer wyjścia, na którym pojawia się sygnał 𝑦𝑚 o maksymalnej wartości. Klasy obiektów zapisane jako wzorce są w postaci wektorów wag 𝑊(𝑚)

Warstwę neuronów, z których każdy ma ten sam zestaw sygnałów wejściowych (wektor sygnałów):   
 𝑋=<𝑥1,𝑥2,…,𝑥𝑛>𝑇  
Ponumerowane neurony mają wektor wag: 𝑊(𝑚)=<𝑤1(𝑚),𝑤2(𝑚),…,𝑤𝑛(𝑚) >𝑇

Sygnał wyjściowy m-tego neurony można wyznaczyć ze wzoru:

𝑦𝑚= 𝑊(𝑚)∗𝑋 =∑ 𝑤𝑖(𝑚)∗𝑥𝑖

1. **Algorytm uczenia: algorytm wstecznej propagacji błędów**

Uczenie metodą wstecznej propagacji błędów jest podstawową metodą uczenia sieci. Algorytm został opracowany w 1974 roku przez P.J. Werbosa jako uczenie z nadzorem lub inaczej - z nauczycielem. Jego nazwa wywodzi się od sposobu obliczania błędu na każdym neuronie w poszczególnych w warstwach sieci.

Algorytm opiera się na sprowadzeniu wartości funkcji błędu sieci poniżej pewnego założonego minimum. Najczęściej jako miarę błędu stosuje się błąd średniokwadratowy na neuronach wyjściowych.

http://www.ai.c-labtech.net/sn/img/image25.gif

http://www.ai.c-labtech.net/sn/img/image26.gif

**Punkty z krokami algorytmu wstecznej propagacji błędów:**

1. Zakładamy że {1,2,3,4,5,6}- neurony
2. Po przetworzeniu sygnału przez całą sieć dostajemy sygnał wyjściowy y. Obliczenie delty dla neuronu 6 jest proste: 𝛿6=𝑧−𝑦
3. Aby obliczyć błędy 𝛿4𝛿5dla neuronów nr 4 , 5 musimy postąpić następująco:

𝛿4=𝑤4,6∗𝛿6

𝛿5=𝑤5,6∗𝛿6

gdzie w5,6 oznacza wagę pomiędzy neuronem 5 i 6 (wagę jeszcze nie skorygowaną)

1. Dla neuronów 1,2,3 :

𝛿1=𝑤1,4∗𝛿4+𝑤1,5∗𝛿5

𝛿2=𝑤2,4∗𝛿4+𝑤2,5∗𝛿5

𝛿3=𝑤3,4∗𝛿4+𝑤3,5∗𝛿5

1. Mając wyznaczone delty dla każdego z neuronów dokonujemy aktualizacji i wag - przykładowo:

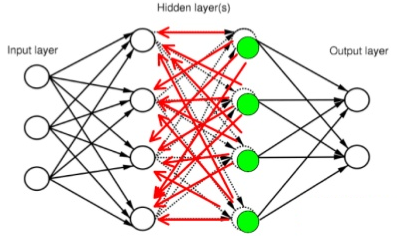
𝑤𝑥1,1′=𝑤𝑥1,1+𝛿1∗𝜂\*𝑑𝑓1(𝑒)/𝑑𝑒∗𝑥1

𝑤3,5′=𝑤3,5+𝛿5∗𝜂\*𝑑𝑓5(𝑒)/𝑑𝑒∗𝑦3

𝛿𝑖 – delta i-tego neuronu

𝑦𝑖 – sygnał z i-tego neuronu warstwy ukrytej lub ostatniej

𝜂 – współczynnik proporcjonalności



**Rys. 1** **Schemat algorytmu wstecznej propagacji błędów**

1. **Reguła uczenia Delta**

Reguła Delta zmienia wagi w sieci proporcjonalnie do błędu na wyjściu sieci (różnicy pomiędzy rzeczywistym sygnałem wyjściowym a sygnałem wzorcowym) i wielkości odpowiedniej sygnału wejściowego. O zachowaniu pojedynczego neuronu decyduje wektor wag W, a o działaniu sieci – macierz wag W’. Aby zapewnić możliwość uczenia trzeba wprowadzić do modelu neuronu dwa dodatkowe elementy: procesor zmiany wag i detektor błędu. Taki neuron nazywa się ADALINE. Sygnał wyjściowy y jest związany z wygnałem wejściowym X zależnością: y =f(X).

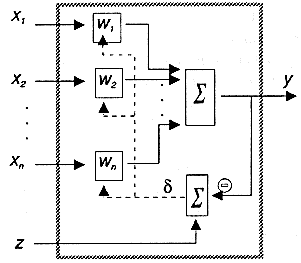
Funkcja f nie musi być zadana w sposób jawny, wystarczy że dla każdego wektora wejściowego potrafimy wskazać konkretną wartość z = f(X) stanowiącą nasze żądane odniesienie sygnału wyjściowego y.

Algorytm reguły Delta, zakłada że wraz z każdym wektorem wejściowym X do neuronu podawany jest sygnał z. Neuron odpowiada na sygnał X sygnałem y = W \* X. Przy czym jeśli neuron nie jest nauczony, sygnał ten jest inny niż wymagany (y≠z). Wewnątrz neuronu istnieje blok oceniający wielkość błędu δ = z – y. Blok ten składa się inwertora oraz sumatora. Na podstawie sygnału błędu oraz wektora wejściowego X możliwe jest takie skorygowanie wektora wag W, by neuron lepiej realizował zadaną funkcję y= f(X). Nowy wektor wag W’ obliczany jest ze wzoru:

W’ = W + ηδX

**Punkty z krokami ogólnego algorytmu reguły Delta:**

1. Mamy dane wektor wejść X {x1,x2 … xn}
2. Dla wejścia X mamy dany wynik odpowiedzi neuronu Y, oraz wymaganą odpowiedź neuronu Z tzn. wraz z danymi wejściowymi X dajemy wymaganą, poprawną odpowiedź neuronu Z
3. Gdy odpowiedź neuronu Y<> Z to oznacza że neuron nie jest nauczony
4. W przypadku gdy wektor nie jest nauczony dokonujemy obliczenia błędu: 𝛿=𝑧−𝑦
5. Po obliczeniu błędu dokonujemy korekty wag: 𝑤′=𝑤+𝜂𝛿𝑥, gdzie 𝜂- współczynnik proporcjonalności.



**Rys.2 Przykładowy schemat ilustrujący regułę Delta**

**Zastosowanie uczenia regułą Delta:**

Duże zastosowanie tego typu układów występuje we wdrażaniu modeli pamięci kojarzeniowej, a także w połączeniu z systemami ekspertowymi (odpowiadanie na pytania użytkowników).

1. **Zestaw danych uczących i testujących**  
   Zbiór danych zawiera 10 dużych i 10 małych liter alfabetu łacińskiego. Są one reprezentowane w postaci dwuwymiarowej tablicy 5x7 pikseli dla jednej litery.

Przykład litery z pliku.txt: 0 1 1 1 0 „A”

1 0 0 0 1

1 0 0 0 1

1 1 1 1 1

1 0 0 0 1

1 0 0 0 1

1 0 0 0 1

Zostały przygotowane 2 ciągi danych: uczący i weryfikujący.

1. **Zestawienie otrzymanych wyników**

Przedstawienie otrzymanych wyników uczenia w postaci tabel i wykresów.

- w zależności od współczynnika uczenia oraz ilości danych uczących

Współczynnik uczenia: 0,1

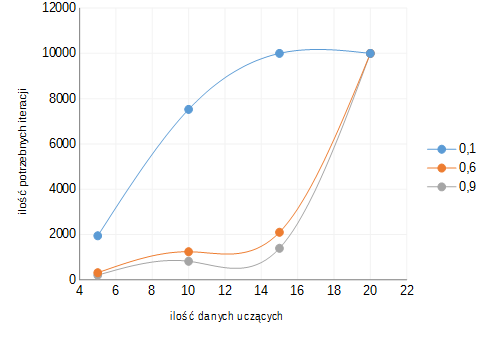
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ilość danych uczących** | **5** | **10** | **15** | **20** |
| **ilość potrzebnych iteracji do nauki sieci** | **1942** | **7526** | **9999** | **9999** |
| **Błąd** | **0,02** | **0,01** | **0,05** | **0,17** |

Współczynnik uczenia: 0,6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ilość danych uczących** | **5** | **10** | **15** | **20** |
| **ilość potrzebnych iteracji do nauki sieci** | **314** | **1241** | **2100** | **9999** |
| **Błąd** | **0,01** | **0,01** | **0,03** | **0,1** |

Współczynnik uczenia: 0,9

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ilość danych uczących** | **5** | **10** | **15** | **20** |
| **ilość potrzebnych iteracji do nauki sieci** | **206** | **820** | **1392** | **9999** |
| **Błąd** | **0,02** | **0,01** | **0,04** | **0,11** |



Im mniejszy współczynnik uczenia, tym większa ilość potrzebnych iteracji do nauki sieci. Najmniejszy błąd podczas testowania zgadywania liter okazał się dla współczynnika 0,6. Okazało się, że największe błędy są dla największej liczby danych do nauki.

1. **Analiza i dyskusja błędów uczenia i testowania w zależności od wartości współczynnika uczenia oraz wybranego algorytmu**

Podczas testowania okazało się, że wybranie małego **współczynnika uczenia** doprowadziło do bardzo powolnego procesu uczenia (wagi były poprawiane w każdym kroku bardzo słabo, żeby więc osiągnęły pożądane wartości trzeba było wykonać bardzo dużo takich kroków).

Z kolei wybór za dużego współczynnika uczenia powodował bardzo gwałtowne zmiany parametrów sieci, które w krańcowym przypadku prowadziły nawet do niestabilności procesu uczenia.

Jednym z błędów uczenia się sieci (**wykorzystanie algorytmu wstecznej propagacji**) była sytuacja kiedy wagi na początku przyjmowały identyczne wartości. Jest to nieprawidłowość, ponieważ rozwiązanie zagadnienia wymaga istnienia wag o różnych wartościach, w przeciwnym razie sieć może się nigdy nie nauczyć. Spowodowane jest to faktem, że rzutowany wstecz błąd warstwy wyjściowej jest proporcjonalny do wag połączeń. Jeśli wagi są sobie równe, to poprawki do wag w danej warstwie też są sobie równe i zróżnicowanie wag nigdy nie nastąpi.

1. **Wnioski**

Po przeprowadzonym eksperymencie można zauważyć, że ważny jest współczynnik uczenia sieci, który powoduje że sieć jest stabilniejsza i popełnia mniej błędów, oraz błąd docelowy do którego zmierza sieć w procesie uczenia. Istotne jest również sposób przygotowania zbioru uczącego, ponieważ nie może zawierać elementów o podobnym wyglądzie a różnych odpowiedziach w procesie uczenia, gdyż sieć jest wówczas źle uczona co powoduje zwiększenie ilości popełnianych błędów. Sieć neuronowa udziela mniej błędnych odpowiedzi kiedy zbiór uczący składa się z uporządkowanych symboli liter niż z przypadkowo ułożonych elementów.

Współczynnik uczenia, zwany learning rate, może być wybierany dowolnie, jednak każda konkretna decyzja ma określone konsekwencje.

Współczynnika uczenia powinien uwzględniać zarówno korzyści związane z szybką pracą, jak i względy bezpieczeństwa, wskazujące na konieczność uzyskania stabilnej pracy procesu uczenia.

Decyduje o wpływie błędu popełnianego przez neuron na korektę wartości wag. Właściwy dobór ma kluczowe znaczenie dla prędkości zbieżności algorytmu. Zbyt mała wartość spowalnia proces uczenia i zwiększa ryzyko wpadnięcia w pułapkę lokalnego minimum (punkt reprezentujący konfiguracje sieci porusza się „małymi kroczkami” na wykresie).

W procesie uczenia sieci ważna jest inicjalizacja początkowego układu wag połączeń między elementami sieci. Najskuteczniejszym rozwiązaniem jest losowy dobór wag, co praktycznie całkowicie zabezpiecza przed problemem z symetrią.

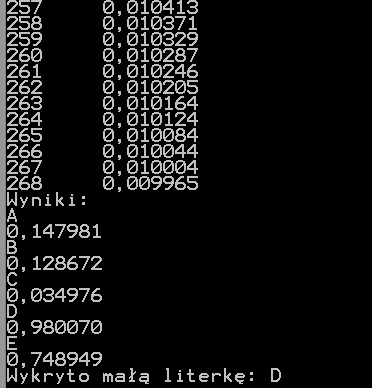
Wybór algorytmu wstecznej propagacji błędów – dzięki zastosowaniu specyficznego sposobu propagowania błędów uczenia sieci powstałych na jej wyjściu, tj. przesyłania ich od warstwy wyjściowej do wejściowej, algorytm propagacji wstecznej stał się jednym z najskuteczniejszych algorytmów uczenia sieci.

**Zastosowania**

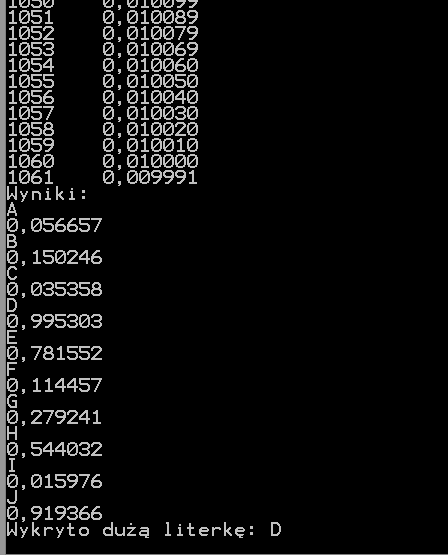
Powyższe testy wykazały że, sieć neuronowa może być wykorzystana dla klasyfikacji i rozpoznawania. Sieć potrafi po „pokazaniu'' jej wystarczającej ilości informacji rozpoznawać w zbiorze danych występujące klasy danych i klasyfikować nowe wzorce według nauczonych wcześniej kategorii. Może to być np. rozpoznawanie pisma, cyfr, obrazów.

**Wybrane wyniki konsoli:**

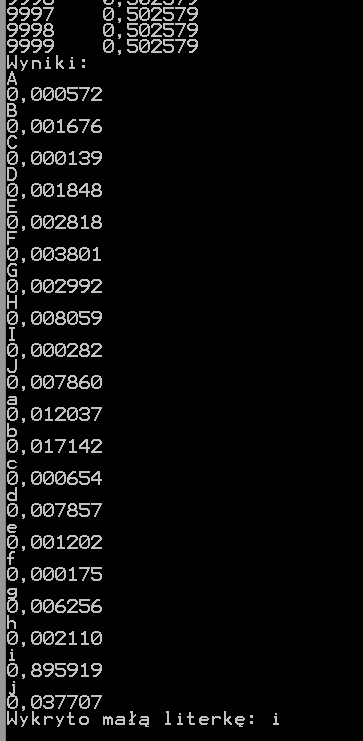
5 danych uczących



10 danych uczących



Cały zestaw danych uczących 20 liter



**Listing kodu**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Laboratoria\_2\_projekt

{

class Program

{

private readonly Siec siec = new Siec();

private readonly string[] dane = {"A", "B","C","D","E","F","G","H","I","J",

"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"};

private static int rozmiarObrazu = 5;

static void Main(string[] args){

var p = new Program();

p.siec.mmacIteracji = int.Parse(10000.ToString());

p.Testowanie();

p.Sprawdzanie();

Console.ReadKey();

}

private void Testowanie() {

int dlugosc = dane.Length;

siec.Inicjacja(rozmiarObrazu \* rozmiarObrazu, dlugosc);

double[][] tab\_wejscie = new double[dlugosc][];

double[][] tab\_wyjscie = new double[dlugosc][];

for (int i = 0; i < dlugosc; ++i) {

tab\_wyjscie[i] = new double[dlugosc];

for (int j = 0; j < dlugosc; ++j) {

tab\_wyjscie[i][j] = i == j ? 1.0 : 0.0;

}

siec.warstwaWyjscia[i].wartosc = dane[i];

}

tab\_wejscie[0] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1};

tab\_wejscie[1] = new double[25] {1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0};

tab\_wejscie[2] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1};

.

.

.

siec.trenujSiec(tab\_wejscie, tab\_wyjscie);

Console.WriteLine("Errors:");

for (int i = 0; i < siec.currentIteration; ++i) {

Console.WriteLine( i + "\t"+ siec.bledy[i].ToString("#0.000000"));

}

}

private void Sprawdzanie(){

double[] przyklad = new double[rozmiarObrazu \* rozmiarObrazu];

przyklad = new double[25] {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0};

siec.Rozpoznaj(przyklad);

Console.WriteLine("Wyniki:");

var wyniki = new double[dane.Length];

for (int i = 0; i < siec.warstwaWyjscia.Length; ++i) {

wyniki[i] = siec.warstwaWyjscia[i].wyjscie;

Console.WriteLine(siec.warstwaWyjscia[i].wartosc);

Console.WriteLine(siec.warstwaWyjscia[i].wyjscie.ToString("#0.000000"));

}

int w = wyniki.ToList().IndexOf(wyniki.Max());

if(w > dane.Length/2)

Console.WriteLine("Zostałą wykryta mała literka: " + siec.warstwaWyjscia[w].wartosc);

else {

Console.WriteLine("Została wykryta duża literka: " + siec.warstwaWyjscia[w].wartosc);

}

}

}

class Siec

{

public struct WarstwaWejscia {

public double wartosc;

public double[] wagi;

}

public struct WarstwaWyjscia {

public double sumaWejsc;

public double wyjscie;

public double blad;

public double Target;

public string wartosc;

}

public double learningRate = 0.6;//współczynnik uczenia

public int rozmiarObrazu = 0;

public int numerWejscia = 0;

public int numerWyjscia = 0;

public WarstwaWejscia[] wartstwaWejscia = null;

public WarstwaWyjscia[] warstwaWyjscia = null;

public double[] bledy = null;

public int currentIteration = 0;

public int mmacIteracji = 10;

public void Inicjacja(int inputSize, int outputSize) {

numerWejscia = inputSize;

numerWyjscia = outputSize;

wartstwaWejscia = new Siec.WarstwaWejscia[inputSize];

warstwaWyjscia = new Siec.WarstwaWyjscia[outputSize];

Random random = new Random();

for (int i = 0; i < numerWejscia; ++i) {

wartstwaWejscia[i].wagi = new double[numerWyjscia];

for (int j = 0; j < numerWyjscia; ++j) {

wartstwaWejscia[i].wagi[j] = random.Next(1, 3) / 100.0;

}

}

}

public bool trenujSiec(double[][] wejscia, double[][] wyjscia)

{

double obecnyBlad = 0.0, maxBlad = 0.01;

currentIteration = 0;

bledy = new double[mmacIteracji];

do {

obecnyBlad = 0;

for (int i = 0; i < wejscia.Length; ++i){

CalculateOutput(wejscia[i], warstwaWyjscia[i].wartosc);

BackPropagation();

obecnyBlad += GetError();

}

bledy[currentIteration] = obecnyBlad;

++currentIteration;

}

while (obecnyBlad > maxBlad && currentIteration < mmacIteracji);

if (currentIteration <= mmacIteracji) {

return true;

}

return false;

}

private void CalculateOutput(double[] pattern, string output)

{

for (int i = 0; i < pattern.Length; i++) {

wartstwaWejscia[i].wartosc = pattern[i];

}

for (int i = 0; i < numerWyjscia; i++) {

double total = 0.0;

for (int j = 0; j < numerWejscia; j++){

total += wartstwaWejscia[j].wartosc \* wartstwaWejscia[j].wagi[i];

}

warstwaWyjscia[i].sumaWejsc = total;

warstwaWyjscia[i].wyjscie = Activation(total);

warstwaWyjscia[i].Target = warstwaWyjscia[i].wartosc.CompareTo(output) == 0 ? 1.0 : 0.0;

warstwaWyjscia[i].blad = (warstwaWyjscia[i].Target - warstwaWyjscia[i].wyjscie) \* (warstwaWyjscia[i].wyjscie) \* (1 - warstwaWyjscia[i].wyjscie);

}

}

private void BackPropagation()

{

for (int j = 0; j < numerWyjscia; j++) {

for (int i = 0; i < numerWejscia; i++){

wartstwaWejscia[i].wagi[j] += learningRate \* (warstwaWyjscia[j].blad) \* wartstwaWejscia[i].wartosc;

}

}

}

private double GetError()

{

double total = 0.0;

for (int j = 0; j < numerWyjscia; j++){

total += Math.Pow((warstwaWyjscia[j].Target - warstwaWyjscia[j].wyjscie), 2.0) / 2.0;

}

return total;

}

private double Activation(double x) {

return (1.0 / (1.0 + Math.Exp(-x)));

}

public void Rozpoznaj(double[] input)

{

for (int i = 0; i < numerWejscia; i++) {

wartstwaWejscia[i].wartosc = input[i];

}

for (int i = 0; i < numerWyjscia; i++) {

double total = 0.0;

for (int j = 0; j < numerWejscia; j++) {

total += wartstwaWejscia[j].wartosc \* wartstwaWejscia[j].wagi[i];

}

warstwaWyjscia[i].sumaWejsc = total;

warstwaWyjscia[i].wyjscie = Activation(total);

}

}

}

}